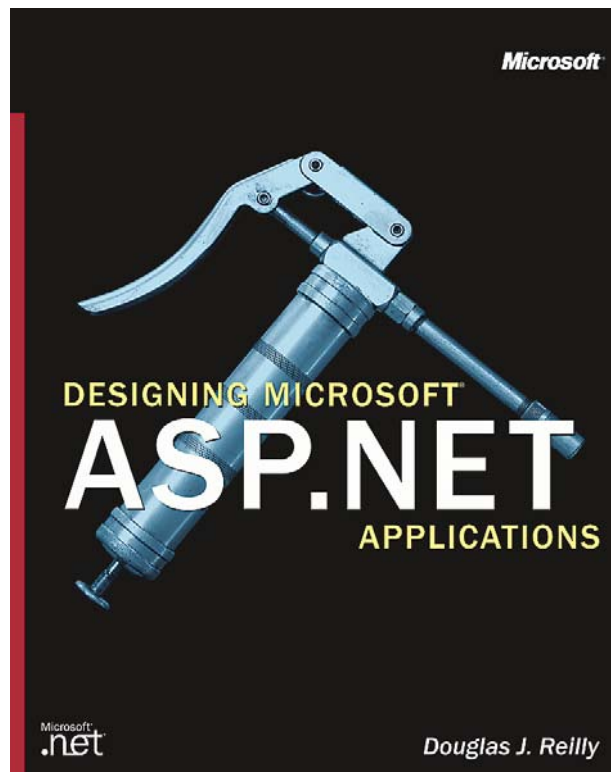


Designing Microsoft® ASP.NET Applications

By

Douglas J. Reilly

ISBN: 0-7356-1348-6



This sample chapter provided courtesy of Microsoft Press.

Microsoft Press produces comprehensive learning solutions that empower home and corporate users, IT professionals, and software developers to do more exciting things with Microsoft technology. Microsoft Press offers hundreds of computer books, interactive training software, and online resources, all designed to help build your skills and knowledge - how, when, and where you learn best.

For more information, go to <http://www.microsoft.com/mspress>

Contents

- 1 Introduction to ASP.NET Development
- 2 Managed Code and the Common Language Runtime
- 3 The .NET Framework Objects and Languages
- 4 ASP.NET Development 101 (Sample Chapter)**
- 5 Web Forms
- 6 Creating ASP.NET Components
- 7 Balancing Server and Client Functionality
- 8 Time to Get the Data
- 9 Data and ASP.NET Forms
- 10 XML Web Services
- Appendix A Configuring ASP.NET Applications in IIS
- Appendix B What You Need to Know About HTML to Use This Book

4



ASP.NET Development 101

Once you've decided that you need to create dynamic Web content (and you certainly will), your next decision is to choose the tools you'll use to develop it. In Chapter 1, you learned about some of the traditional options, such as the Common Gateway Interface (CGI), the Internet Server Application Programmers Interface (ISAPI), and Active Server Pages (ASP). ASP.NET is the newest tool for developing dynamic Web applications, and in this chapter, you'll find out what you need to know to start using it. ASP.NET development has much in common with traditional ASP development, but it also has many differences. Throughout the chapter, you'll see special notes that I've added to highlight the differences between ASP and ASP.NET for you ASP programming veterans.

Hello ASP.NET World!

As with all discussions of programming languages, it's almost mandatory that the first example presented be of the "Hello World" variety. The brief introduction to ASP in Chapter 1 showed the typical "Hello World" example. This example is reproduced in Listing 4-1.

```
<% Option Explicit %>
<HTML>
<HEAD>
<TITLE>Hello ASP World</TITLE>
</HEAD>
<BODY>
<CENTER>
<%
Dim x
```

Listing 4-1 SayHelloASP.asp sample application listing

(continued)

Listing 4-1 *continued*

```

For x=1 to 5
    Response.Write("<FONT size=" & x)
    Response.Write(">Hello ASP WorlD</FONT><BR>" & vbCrLf)
Next
%>
</CENTER>
</BODY>
</HTML>

```

A C# Example

The source code to produce basically the same result, written for ASP.NET, is shown in Listing 4-2.

```

<%@ Page Language="C#" %>

<HTML>
<HEAD>
<TITLE>
My First ASPX Page
</TITLE>
</HEAD>
<BODY>
<CENTER>
<%

int loop;
String s="";
for ( loop=1 ; loop<=5 ; loop++ )
{
    s=s +
        String.Format(
            "<FONT SIZE={0}>Hello ASP.NET WorlD</FONT><BR>",
            loop);
}
Message.InnerHtml=s;
%>
<SPAN id="Message" runat=server/>

</CENTER>
</BODY>
</HTML>

```

Listing 4-2 SayHelloASPDOTNET.aspx sample application listing



ASP.NET Differences ASP file names have the extension .asp. ASP.NET file names generally have the extension .aspx. (Other extensions are associated with ASP.NET, but the rough equivalent of .asp for ASP.NET applications is .aspx.) ASP and ASP.NET files can coexist side by side on a Web site; however, they won't share common application settings or session information. It's generally better to have ASP and ASP.NET applications in separate directories, interacting only via standard arguments on URLs or through a common database.

As you go through Listing 4-2, you'll notice lots of differences between the ASP.NET version and the ASP version in Listing 4-1. The first difference is the language used. Rather than Visual Basic Scripting Edition (VBScript), or even any version of Visual Basic, the page uses C#. In the first line, rather than the *Option Explicit* directive in the ASP example, SayHelloASPDOTNET.aspx uses a *Page* directive that also specifies the language to be used—in this case, C#. Because the page uses C#, *Option Explicit* isn't required to force declaration of variables, and in any event, it isn't supported.

The *Page* directive has several attributes that are important to know about. A partial list is shown in Table 4-1.

Table 4-1 Attributes of the *Page* Directive

Attribute	Description
<i>Buffer</i>	Indicates whether HTTP response buffering is enabled. If <i>true</i> (the default), page buffering is enabled; if <i>false</i> , buffering is not enabled. Generally, buffering a page until all content is written can improve performance of the page, although it can make the page appear to be slower because nothing is visible on the browser until all content is written. Complex pages often exist within HTML tables. If an entire page is contained within a table, it won't be rendered until the table is closed out, so explicit buffering when content is within a table has no downside.
<i>ContentType</i>	Defines the HTTP content type of the response as a standard Multipurpose Internet Mail Extensions (MIME) type. For instance, if this attribute is set to <i>Application/MSWord</i> , the application associated with the .doc file type will be called to open the document, instead of the standard HTML browser.
<i>EnableSessionState</i>	If <i>true</i> , enables session state; if <i>ReadOnly</i> , allows reading but not writing of session state; and if <i>false</i> , disables session state.

(continued)

Table 4-1 *continued*

<i>EnableViewState</i>	Indicates (<i>true</i> or <i>false</i>) whether view state is maintained across page requests.
<i>ErrorPage</i>	Sets a target URL for redirection if an unhandled error occurs.
<i>Explicit</i>	If set to <i>true</i> , the <i>Option Explicit</i> mode in Visual Basic .NET is enabled, requiring variables be declared.
<i>Inherits</i>	Defines a code-behind class for the page to inherit. This attribute can be any class derived from the <i>Page</i> class.
<i>Language</i>	Specifies the language used in all inline code blocks (enclosed in <code><% %></code>). This attribute can be any .NET supported language, including Visual Basic, C#, or JScript .NET.
<i>Strict</i>	If set to <i>true</i> , the <i>Option Strict</i> mode in Visual Basic .NET is enabled, forcing the type of variable to be declared and disallowing narrowing type conversions.
<i>Trace</i>	Indicates (<i>true</i> or <i>false</i>) whether tracing is enabled. The default setting is <i>False</i> . This attribute is used for debugging.
<i>Transaction</i>	Indicates whether transactions are supported on the page. This attribute can be set to <i>NotSupported</i> , <i>Supported</i> , <i>Required</i> , or <i>RequiresNew</i> .
<i>WarningLevel</i>	Indicates the warning level at which the compiler should abort compilation of a page. This attribute can be set to 0 through 4.

There can be only one page directive per .aspx page. After the *Page* directive, the next few lines of SayHelloASPDOTNET.aspx are fairly standard HTML. Inside the code block (between the opening `<%` and the closing `%>`), you can see that the language used is C#. C/C++ programmers should be comfortable with the code that follows within the script block.

Two variables are declared, an integer named *loop* and a string named *s*. The *for* loop steps through the font sizes between 1 and 5. I use the variable *s* to hold all the HTML code to be written out. After the loop, rather than using *Response.Write* to write the output (a *very* ASP way of doing things), I set the *InnerHtml* property of the *Message* object. *Message* is an HTML `<SPAN>` tag declared farther down in the actual HTML, as follows:

```
<SPAN id="Message" runat=server/>
```



ASP.NET Differences If you tried to use *Response.Write* to write the text in this example, it wouldn't compile. Strictly speaking, this isn't a limitation of ASP.NET, but rather a difference between Visual Basic and C#. Visual Basic isn't case-sensitive; C# is.

Notice that the *id* attribute of the *SPAN* element is what is used to reference the HTML object. One other important feature of the *SPAN* element is that it has a *runat* attribute, indicating that it should be run at the server. Running controls on the server is common in ASP.NET applications. Also notice that there's no closing `</SPAN>` tag. This seeming omission isn't a problem because the trailing `/>` in the tag indicates that it's a start tag and an end tag.

Note Even if you explicitly place text within the `<SPAN id="Message" runat=server>` tag and a `</SPAN>` end tag, the text won't be rendered in the browser because the line `Message.InnerHtml=s;` resets the text to the string in the variable `s`. If instead you changed that line to read `Message.InnerHtml=Message.InnerHtml + s;`, the text in the actual HTML between the `<SPAN>`/`</SPAN>` tags would be displayed, followed by the text created in the loop.

A Visual Basic .NET Example

The same page created using Visual Basic .NET is shown in Listing 4-3. This listing isn't much different from the C# example presented in Listing 4-2.

```
<%@ Page Language="VB" %>

<HTML>
<HEAD>
<TITLE>
My First ASPX Page
</TITLE>
</HEAD>
<BODY>
<CENTER>
<%

Dim tLoop as Integer
Dim s as String
s=""
For tLoop=1 to 5
    s= s + String.Format( _
        "<FONT SIZE={0}>Hello ASP.NET World</FONT><BR>", _
        tLoop)
```

Listing 4-3 SayHelloASPDOTNETVB.aspx sample application listing

(continued)

Listing 4-3 *continued*

```

Next
Message.InnerHtml=s
%>
<SPAN id="Message" runat=server />

</CENTER>
</BODY>
</HTML>

```

Figure 4-1 shows the output from the Visual Basic .NET version of the page; however, both the Visual Basic .NET and the C# versions produce nearly identical output.

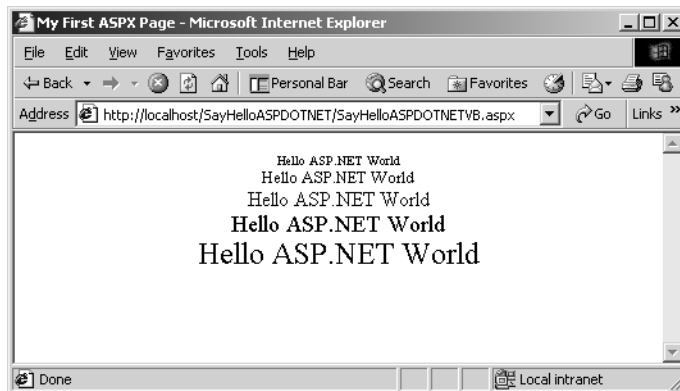


Figure 4-1 Output of the Visual Basic .NET version of the ASP.NET example shown in Listing 4-3

I've made some obvious syntax changes in this example, such as removing semicolons at the ends of statements, changing curly braces ({ and }) and the *for* loop syntax, as well as changing variable declarations. In addition, although I commonly use a variable named *loop* in C/C++ and now in C#, *loop* is a reserved word in Visual Basic .NET—therefore, I changed the variable name.

The ASP.NET Development Model

Developing applications using ASP.NET is similar to developing applications using earlier versions of ASP, as far as the overall development model is concerned. Figure 4-2 shows the general workflow from the ASP developer's perspective. The process consists of editing the page, testing the page, possibly returning to edit the page again, and so on.

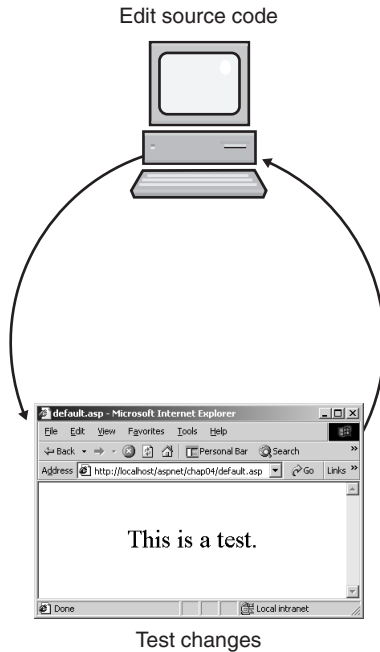


Figure 4-2 The ASP development cycle

For the ASP.NET developer, the process is exactly the same, but the underlying activity is slightly different. Figure 4-3 shows the real process, which is edit-compile-test. The compile portion of the process is transparent to the developer. Every time an application is run, if the page being run hasn't been compiled, it's compiled automatically.

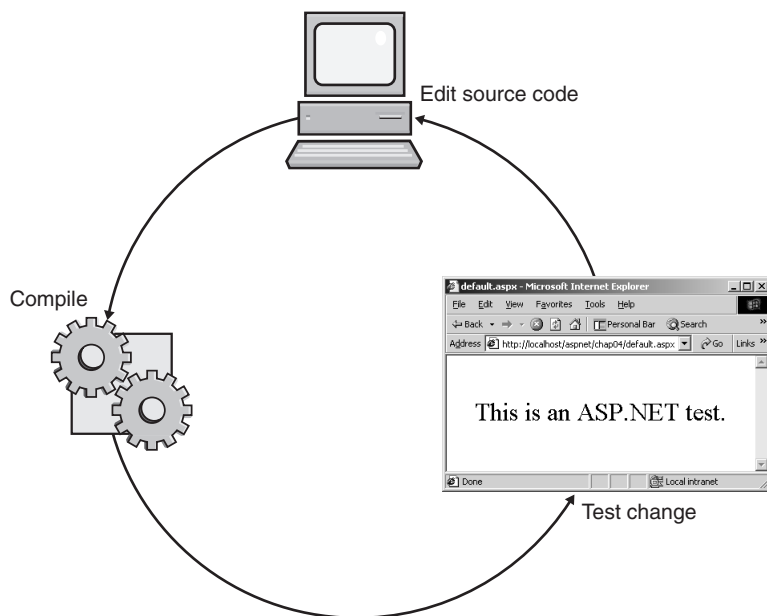


Figure 4-3 The ASP.NET development cycle

What's interesting about the ASP.NET development cycle is that while the developer's impression is that nothing has changed in moving from ASP to ASP.NET, the change under the covers has a tremendous impact. The primary benefits, aside from the new languages and the .NET Framework, are that ASP.NET applications offer better performance than ASP and greater reliability, since compiling the application ensures that blatant syntax errors are caught on the first compilation, instead of being discovered only when the actual code is run.

Creating an ASP.NET Web Application with Visual Studio .NET

Although the ASP.NET Web Application is just one of the types of applications that are possible with ASP.NET, it's the type of application you'll create most often. You don't have to use Visual Studio .NET to create an ASP.NET Web application, but your life as a developer will sure be easier if you do.

When you start Visual Studio .NET, the Start Page is the first screen that appears. The Start Page is designed to introduce you to the Visual Studio .NET environment as well as to allow you to perform many common tasks easily. One of the nicer features of the Start Page is the My Profile option. When this option is selected, you'll see a page similar to the screen shown in Figure 4-4.

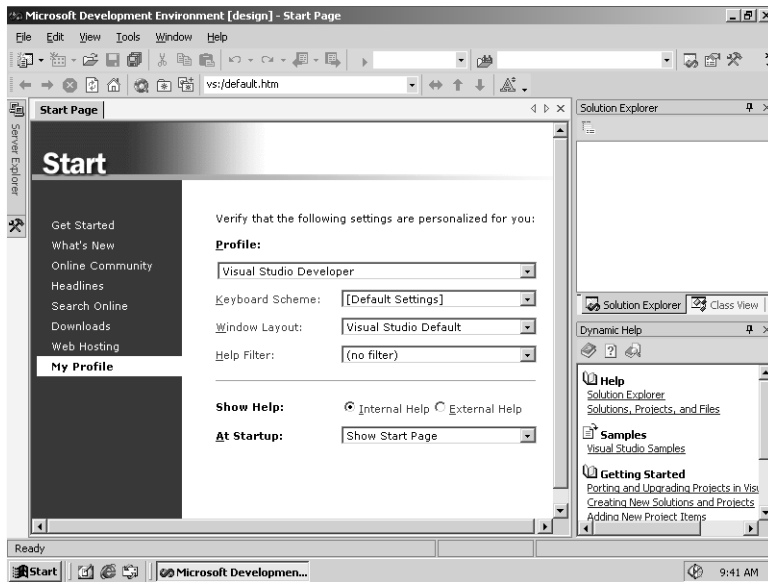


Figure 4-4 The Visual Studio .NET My Profile screen

Many developers moving to Visual Studio .NET will be coming from one of three integrated development environments (IDEs): Visual InterDev, Visual Basic, or Visual C++. Historically, these IDEs have been quite different, and developers who work in one IDE often have strong feelings about the other IDEs. To allow each developer to feel more comfortable with the new, common IDE, Visual Studio .NET allows you to configure different screen layouts and keyboard schemes to resemble what you're used to. Of course, not everyone will be happy with all the decisions the designers of the new IDE made, but having a common IDE is a necessary step toward supporting multilanguage development.

Note Although the IDE supports creating applications in Visual Basic .NET, C#, or C++, it doesn't currently allow you to create a single ASP.NET application with both Visual Basic .NET and C# pages in any clean way. I would hope that future versions of the IDE will allow greater integration of different languages within the same solution. The term *solution*, by the way, is the Visual Studio .NET term used to describe a container that can group multiple related projects.

Changing the screen layouts to emulate the various Visual Studio 6.0 IDEs is an interesting exercise and can give you a feeling of *déjà vu*. Although I've been quite fond of the Visual C++ 6.0 layout, for the examples in the book, I've used the Visual Studio Default layout. After months of use in one form or another, I find the default layout to be quite good.

Once Visual Studio .NET is open, you can create a new project in a variety of ways. The most general way is to click the File menu, point to New, and then click Project. Doing this displays the dialog box shown in Figure 4-5.

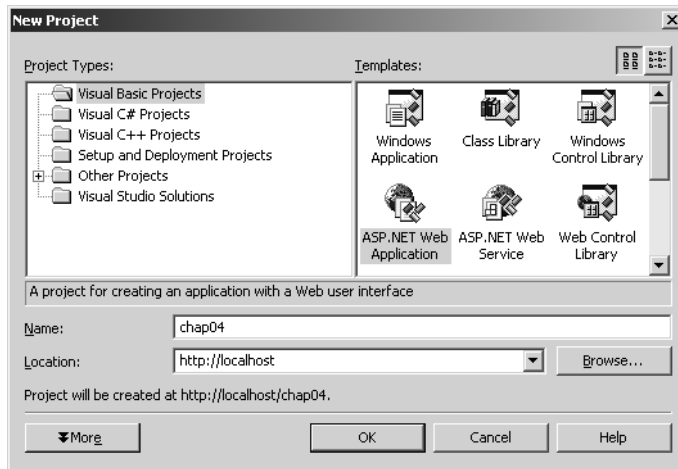


Figure 4-5 The Visual Studio .NET New Project dialog box

The folder view on the left enables you to select the language you want to use for the project or one of several types of special projects. Most often, you'll create a project in one of the language folders. In the Other Projects folder, if you're using the Enterprise edition of Visual Studio .NET, you'll see enterprise template projects that can be useful when you're creating larger distributed applications.

Depending on the type of project you select, the Location text box in the New Project dialog box might change to a folder location or a URL of the local Web server. In Figure 4-5, the location is a virtual directory on the root of the current Web directory because the project type selected is ASP.NET Web Application.

Visual Studio .NET Interactions with Internet Information Services (IIS)

For this example, I'll select the Visual Basic ASP.NET Web Application and name the project chap04. When I click OK, several things happen. As with Visual C++ 6.0, the name of the project becomes the name of the directory where the application is stored. In addition, Visual Studio .NET contacts the Web server (in this example, the local Web server) and creates an application directory by the same name. After the project is created, when I look at the Internet Information Services console, I see the application directory created, as shown in Figure 4-6.

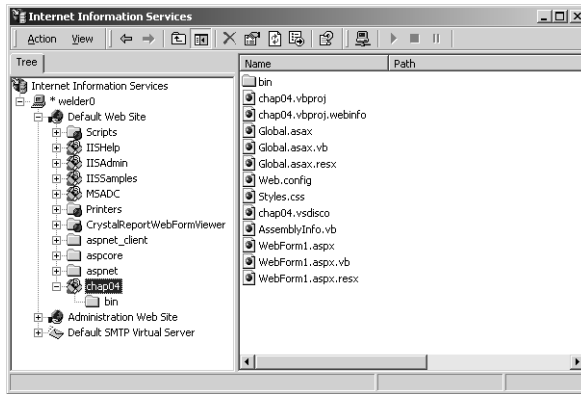


Figure 4-6 The Web application directory created by Visual Studio .NET when a new ASP.NET application is created

The right pane lists all the files created by Visual Studio .NET. The most significant files for you as the developer are the Web Form file (cleverly named WebForm1.aspx) and the code-behind file (named WebForm1.aspx.vb). If this were a C# project, the code-behind file would be named WebForm1.aspx.cs. You can use the Web.config file to customize the application settings, as I'll discuss shortly. Visual Studio .NET also creates a bin folder, in which all the compiled code for the application is stored.

Looking at the properties of the application directory (right-click on the chap04 application package icon and select Properties), you see nothing that unusual. From the Properties page, you can click the Configuration button to display the Application Configuration dialog box shown in Figure 4-7.

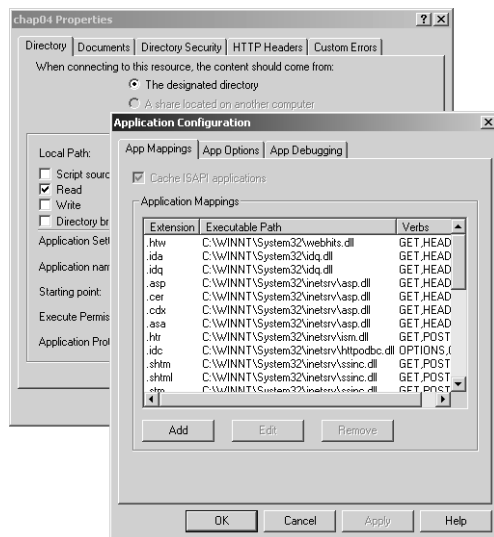


Figure 4-7 Application Configuration dialog box for newly created chap04 application directory

The App Mappings tab displays the executable or DLL that will process a given extension. In this case, the entire executable path is too large to be seen in the dialog box, but you can take my word for it that all ASP.NET extensions are mapped in IIS to C:\WINNT\Microsoft.NET\Framework\v1.0.2941\aspnet_isapi.dll. As I write this, I'm using version 1.0.2941 of the .NET Framework, so you can see that the version is part of the path to the DLL that handles ASP.NET applications. The significance of the long path that includes the version number is that it should be possible to have different ASP.NET applications using different versions of ASP.NET.

Your First Visual Studio .NET Web Page

Once Visual Studio .NET has created the project files and the application directory in IIS, Visual Studio will look something like Figure 4-8. A couple of things are significant about Visual Studio. First, notice the faint grid on the WebForm1.aspx tab. These lines are shown when *Grid Layout* is enabled. Grid Layout allows you to place components precisely, as you would on a traditional Visual Basic form.

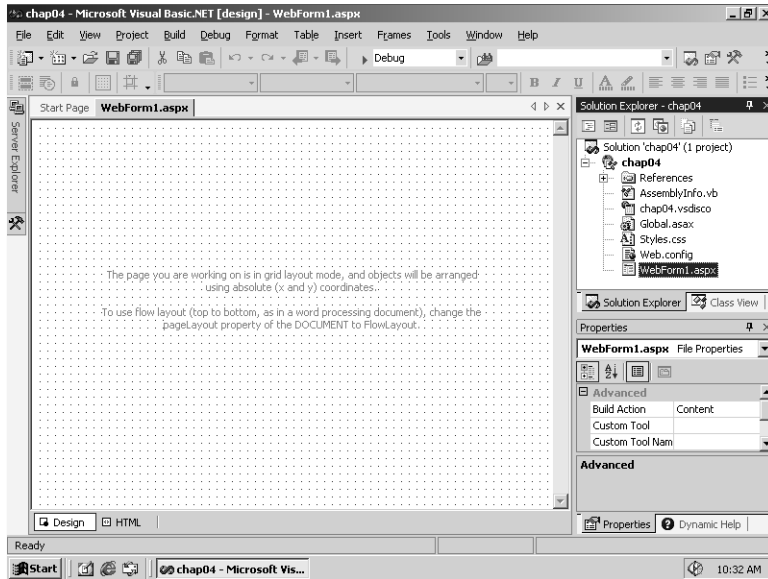


Figure 4-8 Visual Studio as soon as the new Web Application project has been created in Grid Layout

Note The way this magic of providing precise component layout works is worth a brief note. Traditionally, HTML hasn't been able to give you such fine control of the exact placement of components within a Web page. When you place a component using Grid Layout, the component is positioned using DHTML and Cascading Style Sheets (CSS) to it to tell the browser exactly where to render it. This idea is very cool but presents two possible problems. The first is what to do about downscale browsers that don't support DHTML and CSS. To allow the illusion of precise placement to continue, a complex set of tables is sent to the browser, doing an acceptable job of placing components in most cases. A second problem is that trying to use such precise control of a page might cause some developers to create layouts that are fragile. For example, if the fonts installed don't *exactly* match the fonts the developer used, the layout will likely be different. The decision is left to the developer: the Page Layout can be changed from Grid Layout to Flow Layout if you don't want that level of control. This setting is in the property dialog box of the page. If you're developing applications for the Internet rather than an intranet, on which you have control over the clients, it might not be reasonable to take advantage of the admittedly convenient Grid Layout. In general, examples in this book won't use Grid Layout but will instead use tables to align components. The next example is an exception because I'm trying to show the IDE, and Grid Layout does show that off quite well.

Display the Toolbox by clicking the Toolbox tab on the left of the screen (just below the Server Explorer) or by clicking the Toolbox button on the toolbar. In this example, I'll add two labels and place them on the design grid, one on the top and one below it, both just about centered. I'll make the lower label a bit wider than the top label. Your screen should look something like the screen shown in Figure 4-9.

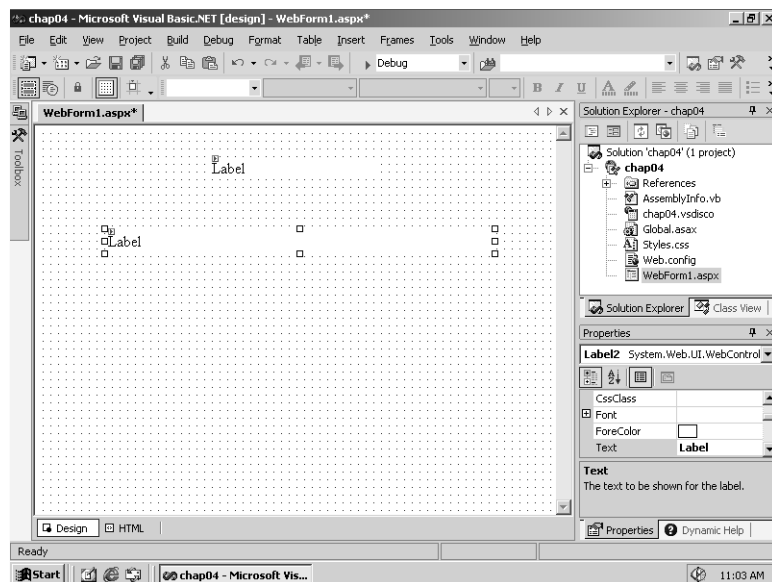


Figure 4-9 The chap04 main form with two labels added to the form

There are two major ways to modify objects on an ASP.NET page at design time. One way is to use the Properties window. This window is by default in the lower right corner of Visual Studio. To change the top label, just click that label (it should be Label1 on the form) and modify the properties. Then change the *Text* property to read, "Your First ASP.NET Page". You may need to resize the label to keep all the text on a single line. Next, go to the *Font* property. This property has a + next to it, meaning that you can expand it to get to subproperties. Change the *Bold* subproperty to *True*. As you make the preceding changes, the changes will show up in the designer immediately.

The second way to modify objects at design time is to change the code. Let's use code to change the other label, *Label2*. You have a couple coding options for changing the text of a label. First, notice the two tabs at the bottom of the design surface: the active tab, Design, and another tab, HTML. Click the HTML tab, and you'll see the HTML code, looking very much like HTML displayed in Visual InterDev 6.0. Figure 4-10 shows what that screen will look like.

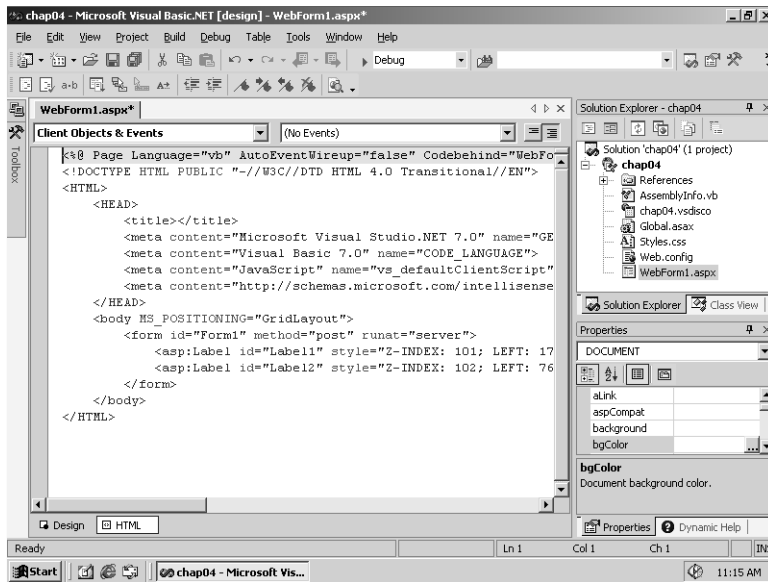


Figure 4-10 HTML code as it appears in Visual Studio .NET

Although it's not visible in the figure, at the very end of the line with the `Label1` tag, the text I entered in the Properties window is between the opening and closing `asp:Label` tags. The *Font-Bold* attribute is also set to *True*, based on the change I made in the Properties window. The designer is a two-way designer; that is, changes made in HTML view also appear on the Design view. For example, if you click the `<body>` opening tag, the Properties window changes to reflect the properties of the body tag. Scroll down in the Properties window to the *bgcolor* property, click on the field, and you can either enter a valid HTML color directly or click the ellipsis button and use the Color Picker dialog box to pick the color. I selected a pale yellow, also known as #ffffcc. The appropriate attribute/value pair is added to the body tag. Now, if you click back to Design view, the background will be the selected color.

Changing text in Design view or HTML view is fine, but you often need to change properties at runtime. To see the Visual Basic code for this page, select Code from the View or simply press the F7 key. The active pane will change to `WebForm1.aspx.vb`, and the Visual Basic .NET code will appear. There is very little code, and some of that is hidden from view, by default. Ignore the hidden code for now. The method that matters is *Page_Load*, which should look like the following (reformatted a bit here for clarity):

```
Private Sub Page_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    'Put user code to initialize the page here
End Sub
```

Rather than putting just static text into Label2, I will put some static text and the current date and time, something that is certain to change each time I refresh. I add the following code just under the wizard comment about placing user code to initialize the page here:

```
Label2.Text = "The current date and time is " + Now()
```

This code is very Visual Basic–like, and it should be clear exactly what I’m doing. Notice that I’m using the plus sign (+) rather than the ampersand (&) for concatenating strings. Use of the plus sign was discouraged in previous versions of Visual Basic but works correctly in both Visual Basic .NET and C#, and so I’ll always use the plus sign to concatenate strings throughout this book.

Once I’ve made all the changes I want to, I can go to the Debug menu and select Start, which will start the application with the debugger. If anything has been changed since the last time the application was run, the affected items will be compiled, so the first time you run the application, it will take longer than normal. If all has gone well, a screen similar to the one shown in Figure 4-11 will appear.

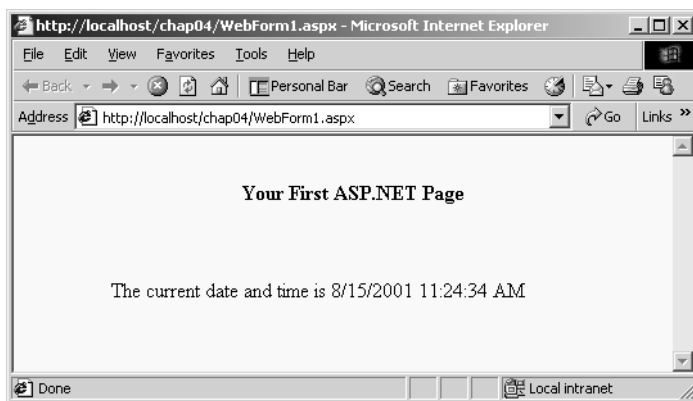


Figure 4-11 The chap04 example page when run after the modifications described in the text

This is a very simple application, but I hope it gives you a feel for some of what is possible in the Visual Studio .NET IDE. Although some of the design features have been available in tools such as Visual InterDev 6.0, the implementation in Visual Studio .NET is much better. I virtually never used the designer in Visual InterDev 6.0 because it had a nasty habit of completely reformatting my nicely formatted HTML. Visual Studio .NET is more intelligent about how it formats your text moving from the designer to the editor, and there are configuration options to control most of the reformatting Visual Studio .NET does.

The server components, such as the Label control, haven't been available before. The label components are barely the tip of the iceberg as far as server controls go. In subsequent chapters, we'll return to developing ASP.NET Web Forms as well as using server controls—even creating your own server controls.

Other Types of ASP.NET Applications

Except for the preceding ASP.NET example, the examples we've looked at so far are very similar to existing ASP applications. A page is requested, and after server-side processing, the HTML code is sent to the browser. If all ASP.NET had to offer was greater efficiency doing what ASP has always allowed, it would still be a great improvement over ASP. But as the late-night infomercial hucksters often say, "But wait, there's more!"

In addition to the ASP-like applications you're already familiar with, you can use ASP.NET to help develop two other types of scalable applications: XML Web services and applications using the HTTP runtime, HTTP handlers, and HTTP modules.

XML Web Services

How often have you had a neat bit of processing that you needed to share with another application, either on an enterprise-wide intranet or over the Internet? For example, suppose you have a bit of code that does some specialized validation, such as a credit card validation function. Given a credit card number, the function returns feedback on whether the card is valid. The function might directly interact with a database, or perhaps it might even interact with some service that has a less than convenient programmer's interface. If you have multiple applications that need to access that functionality, you've had a few ways to make the functionality available.

One way would have been to create a service application that would communicate with the various users of the function via TCP/IP, using a custom protocol. This option isn't terrible, but it does lead to a Tower of Babel of interfaces and protocols. Does the system expect the credit card confirmation result in uppercase ("YES" or "NO") or lowercase ("yes" or "no")? Does the system use commas to delimit the confirmation result from the authorization code, or tildes (~)? What port is it on? Will it work through firewalls? Will anyone remember this a year from now?

The second option has been to create a Web page that, given arguments appended on the URL, will produce a result and return it as a page that can be read by the requesting application rather than displayed in the browser. This option does resolve the problem of making the function available through firewalls, but it does nothing to address the problem of a custom interface that can be easily forgotten.

The solution is XML Web services. Briefly, XML Web services are software components that provide services to applications over the Web and use Extensible Markup Language (XML) for sending and receiving messages (I'll cover XML Web services in greater detail in Chapter 10.) XML Web services are not dependent on the .NET Framework. As a matter of fact, XML Web services don't even require a Windows operating system on the server, and they can be created using any tool that can create a Simple Object Access Protocol (SOAP) compliant application. (MSDN contains an article, "Develop a Web Service: Up and Running with the SOAP Toolkit for Visual Studio" that describes creating an XML Web service using the SOAP Toolkit in conjunction with Visual Studio 6.0; see <http://msdn.microsoft.com/library/periodic/period00/webservice.htm>.)

So why are XML Web services in ASP.NET such a big deal? The reason is the simplicity ASP.NET brings to creating them.

Note XML Web services will revolutionize the way services are available on the Web. For example, at the time of this writing, Microsoft and eBay had announced an agreement to use XML Web services to integrate Microsoft services, such as Carpoint, bCentral, and WebTV, with eBay's marketplace. The kind of cooperation planned would be difficult without using XML Web services.

How simple is it to use the .NET Framework to create XML Web services? Listing 4-4 shows the XML Web service code required in Visual Basic .NET, including the code generated by Visual Studio .NET. The generated code appears between the *#Region* and *#End Region* tags.

```
Imports System.Web.Services

Public Class Service1
    Inherits System.Web.Services.WebService

#Region " Web Services Designer Generated Code "

    Public Sub New()
        MyBase.New()

        'This call is required by the Web Services Designer.
        InitializeComponent()

        'Add your own initialization code after the
        'InitializeComponent() call

    End Sub

    ' Required by the Web Services Designer
    Private components As System.ComponentModel.Container

    'NOTE: The following procedure is required by the Web Services Designer
    'It can be modified using the Web Services Designer.
    'Do not modify it using the code editor.
    <System.Diagnostics.DebuggerStepThrough()> _
        Private Sub InitializeComponent()
            components = New System.ComponentModel.Container()
        End Sub

    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        'CODEGEN: This procedure is required by the Web Services Designer
        'Do not modify it using the code editor.
    End Sub

#End Region

    ' WEB SERVICE EXAMPLE
    ' The HelloWorld() example service returns the string Hello World.
    ' To build, uncomment the following lines then save and build the project.
    ' To test this web service, ensure that the .asmx file is the start page
    ' and press F5.
    ,

    <WebMethod()> Public Function HelloWorld() As String
        HelloWorld = "Hello World"
    End Function

End Class
```

Listing 4-4 Simple HelloWorld XML Web service code

Figure 4-12 shows the results of calling the XML Web service—the XML message that is the string returned by the *HelloWorld* method of the class *Service1*. Of course, the more interesting case is when parameters are passed into the service and the service does something interesting with them to generate results.

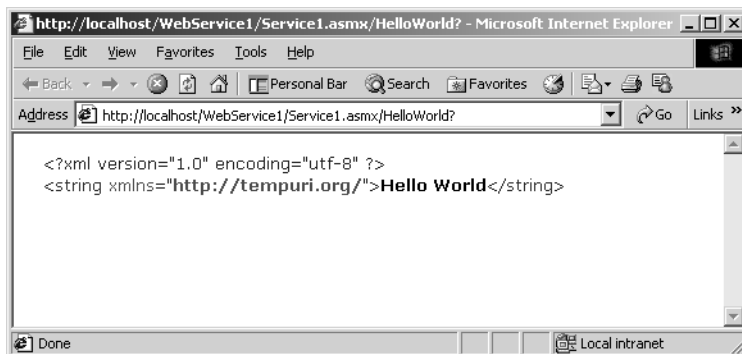


Figure 4-12 The results returned by calling the HelloWorld XML Web service shown in Listing 4-4

XML Web services open up a new world for application developers and let even the smallest development firm expose services over the Web that will create entirely new market opportunities.

HTTP Handlers and HTTP Modules

Additional types of ASP.NET applications are HTTP handlers and HTTP modules. These applications are roughly equivalent to ISAPI extensions and ISAPI filters, respectively. In the ASP world, you would resort to ISAPI extensions or filters under the following two circumstances:

- You hit a performance or scalability wall.
- You need the flexibility offered only by ISAPI extensions or filters.

The first condition isn't likely to be an issue with the current ASP.NET applications that you can develop. Given that ASP.NET applications are compiled using the same runtime as HTTP handlers and HTTP modules, performance and scalability aren't likely to be an issue.

The second condition is much more likely to persist—thus the need for HTTP handlers and HTTP modules. HTTP handlers are helpful if, for example, you need to port an existing CGI application to ASP.NET or if you need to do somewhat unusual things, like return binary data. HTTP modules can act like the binary equivalent of the *Global.asax* file, catching various events, and giving the application developer the greatest flexibility.

Configuring an Application

One element seems to be missing from the Visual Basic .NET code shown in Listing 4-3. Although I've often extolled the virtue of using *Option Explicit*, I didn't use it in this example, which can be done by setting *Explicit*="true" in *@ Page*. I can assure you that the page does require variables to be declared—I missed one of the references to *loop* when converting the application from C# to Visual Basic .NET and an error page did in fact appear, as shown in Figure 4-13.

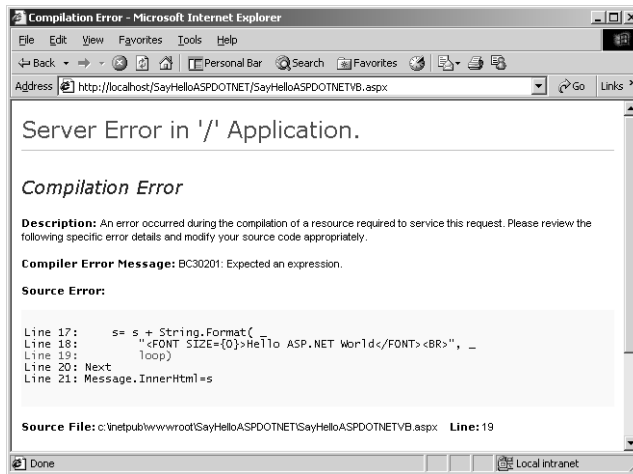


Figure 4-13 The error message that appears when a variable is not declared in the source from Listing 4-3

This error message contains quite a bit more information than ASP error messages had, and it includes the section of code that produced the error, with the line in which the error occurred displayed in red. By default, this detailed error message will appear only on the machine on which the application is running. This is good default behavior because it's possible that the source code displayed could include information about database user names and passwords, or worse.

So if I didn't set *Explicit* to true, why did I get the error message about the undeclared variable (badly reported in this example as "Expected an expression")? The answer is the Web.config file. In addition to specifying *Explicit* and *Strict* on each page, Web.config provides an application-global way to configure these and other application settings. Listing 4-5 shows a simple Web.config file.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
    <compilation
      debug="true"
      defaultLanguage="C#"
      explicit="true"
      batch="true"
      batchTimeout="30"
      strict="true" >
    </compilation>
  </system.web>
</configuration>
```

Listing 4-5 Simple Web.config file, with *explicit* and *strict* set to “true”, eliminating the need for setting these on each Visual Basic .NET page

If you’re familiar with XML, you’ll recognize Listing 4-5 as a simple, well-formed XML document. In Chapter 8, we’ll discuss XML, but in the meantime, you’ll need to be aware of the following aspects of XML:

- XML always uses matched start and end tags, unlike HTML, which allows omission of many end tags. For some tags, the start and end tags can be represented as a single tag. For example, the following tag is valid:

```
<compilation debug="true" />
```

- XML is case-sensitive. Thus, the following tag is *not* valid, because `</Compilation>` is not seen as a closing tag for `<compilation>`:

```
<compilation debug="true"> </Compilation>
```

Although early betas of the .NET Framework were forgiving about the case of attributes values (for example, *true* and *True* were considered the same), from Beta 2 onward, the configuration files are, correctly, completely case-sensitive.

- XML values are enclosed in quotation marks. Thus, the following tag is not valid:

```
<compilation debug=true />
```

XML is the preferred data language of the .NET Framework. As of this writing, however, no automated administrative tool is available for editing the Web.config file. This isn’t a terrible problem because XML is an easy format to follow, but it does mean that you’ll need to manually tweak these files using a normal editor program such as Notepad (known affectionately as Notepad.NET to some of the early ASP.NET adopters). I won’t cover all the many possible

configuration options because they're described in the .NET Framework Software Development Kit (SDK) documentation; however, I will go through the important configuration options and their implications and explain each of the sections of the Web.config file.

Where Does the Web.config File Go?

One of the frustrations for ASP programmers is the odd patchwork of files that grows up around any complex Web site. With ASP, the only configuration file was Global.asa, and in fact, this file is similar to the ASP.NET Global.asax file. ASP beginners often ask, "Where do I put the Global.asa file?" It turns out that in practice you had to put a Global.asa file in almost every directory.

In ASP.NET, the Web.config file provides a mechanism that should allow many sites to have far fewer redundant configuration settings within each virtual site. There is a root configuration file, in the same format as the Web.config files, named Machine.config. This file is included with the .NET Framework and contains many default settings. It is located under the Windows root, in the %windir%\Microsoft.NET\Framework\<version>\CONFIG folder. All other directories on the site inherit settings from this root file and from all Web.config files that exist higher in the logical hierarchy.

For example, one possible elementsection in the Web.config file is *appSettings*. This section is normally used to make certain variables available to all pages within an application, to multiple applications (if the variable exists in a virtual directory with other applications located logically under it), or even to all applications on the machine (if the *appSettings* section is located in the Machine.config file). Individual *appSettings* values can be overridden based on the location of the Web.config file in the hierarchy. Suppose, for example, that Machine.config contains the following section (within the <configuration></configuration> tags):

```
<appSettings>
  <add key="dsn" value="myDSN" />
</appSettings>
```

Suppose further that there is a virtual directory named Test that has the following section within the <configuration></configuration> tags:

```
<appSettings>
  <add key="dsn" value="myLocalDSN" />
</appSettings>
```

If these are the only *appSettings* sections on the machine, any page that retrieves the "dsn" key from *appSettings* will receive the value "myDSN", *except* any page within the Test application. Pages within the Test application or directories logically located below Test will receive the value "myLocalDSN".

Caution If hacking in the registry isn't enough fun for you and you're looking for a new way to hose your machine, do try improperly nesting sections of the Web.config or Machine.config files. Although this will mess up only your ASP.NET applications, it will *thoroughly* mess them up. It's possible that future versions of ASP.NET will be more forgiving of such errors, but the current version is not. Starting with Beta 2 of ASP.NET, all Web.config files are also case-sensitive. This requirement is reasonable from the standpoint that a well-formed XML file is case-sensitive, and the Web.config files are designed to be well-formed XML files. That said, the need for case-sensitivity can still be a pain.

Don't Try This at Home!

Imagine, if you will, a resource that's physically located at c:\Subdir1\Subdir2\Resource.aspx. VirtualDirectory1 is mapped to c:\SubDir1, and VirtualDirectory2 is mapped to c:\Subdir1\Subdir2. If you access Resource.aspx via http://localhost/VirtualDirectory1/Subdir2/Resource.aspx, you could access the file with completely different settings than if you used http://localhost/VirtualDirectory2/Resource.aspx. You can do this because the inheritance of configuration information from Web.config isn't based on the physical directory hierarchy but rather on the logical hierarchy defined by the virtual directory structure.

Obviously, avoiding this kind of setup is important to ensure that all access to a resource uses the same set of configuration settings.

The configuration files contain many sections. What follows in this section is an alphabetic listing and description of the significant sections, along with an example here and there to clarify things as needed.

The *authentication* Section

ASP.NET allows you to authenticate users in a number of ways. An example of the *authentication* section of the Web.config file, set up for forms-based authentication, is shown here:

```
<authentication mode="Forms">
  <forms name=".ASPXUSERDEMO" loginUrl="login.aspx"
    protection="All" timeout="60" />
</authentication>
```

The options for the *mode* attribute of the `<authentication>` tag are listed in Table 4-2.

Table 4-2 Options of the *mode* Attribute

Option	Description
<i>Forms</i>	Uses a user-provided form to gather identifying information
<i>Windows</i>	Uses Windows authentication to obtain the identity of the user
<i>Passport</i>	Uses Microsoft Passport authentication
<i>None</i>	Uses no authentication

Windows authentication in ASP.NET is similar to Windows authentication in earlier versions of ASP. Windows authentication generally piggybacks on IIS support for authentication using the Windows user database. One addition is the use of Windows authentication in addition to specific user and role authorization, as discussed in the next section.

Passport authentication uses an external user database. Computers using Passport authentication must have the Passport SDK installed. ASP.NET provides a wrapper around the Passport SDK.

Forms-based authentication is commonly used for Internet applications, where it's likely that not all users will be members of a Windows domain. Although this type of authentication can be implemented using traditional ASP, ASP.NET makes forms-based authentication much easier by creating a formalized framework to support it.



ASP.NET Differences In ASP, the standard method for conducting forms-based authentication is to use the *Session_OnStart* event handler, placed in *Global.asa* to redirect new sessions to a login page. This method doesn't scale well because the session state in ASP can't be maintained across a Web server farm. The forms-based authentication method provides a cleaner way to ensure that users are logged in.

Note When most of the attributes within all the configuration files are specified, they are specified using *camel casing*, meaning that the initial letter of the attribute name is lowercase but the initial letter of embedded words are capitalized, for example *loginUrl*. This convention is different from some earlier public betas, in which the same attribute might have been specified using *Pascal casing*, resulting in *LoginUrl*.

When forms-based authentication is specified, the `<forms>` subtag can be used. The `<forms>` tag has the attributes listed in Table 4-3.

Table 4-3 Attributes of the `<forms>` Tag

Attribute	Description
<i>loginUrl</i>	The URL to which unauthenticated users are redirected. This URL can be on the same machine or on a different machine, but if it's on a different machine, the <i>decryptionKey</i> attribute must be the same for both machines. <i>decryptionKey</i> is an attribute of the <code><machineKey></code> tag in Machine.config.
<i>name</i>	The name of the cookie to use for authentication purposes. If more than a single application on the machine uses forms-based authentication, the cookie name should be different for each application. ASP.NET uses / as the path of the cookie.
<i>timeout</i>	The number of minutes for expiration of the cookie. The cookie will be refreshed if half the <i>timeout</i> number of minutes has elapsed—an effort to reduce the number of warnings users will get about receiving cookies, if they have cookie warnings turned on. Because cookies can be refreshed, the timeout value might lose precision. Thus, you can't absolutely depend on a cookie timing out in exactly the number of seconds specified by the timeout attribute. The default value is 30.
<i>path</i>	The path for cookies. Defaults to /. This attribute can be changed by specifying a value in the <code><forms></code> tag or can be changed programmatically.
<i>protection</i>	The type of cookie protection. Allowed values are <i>Validation</i> , <i>Encryption</i> , <i>None</i> , and <i>All</i> . <i>Validation</i> validates the cookie data but doesn't encrypt it. <i>Encryption</i> encrypts the cookie data but doesn't validate it. <i>None</i> does neither. <i>All</i> (the default) both encrypts the cookie data and validates it, detecting any alteration in transit. For all but the least important data, the default is a reasonable choice, at the cost of some performance.

Note Why validate the cookie? Because the cookie can be used to tie into information that shouldn't be shared, validating the cookie data and rejecting it if it has been tampered with can ensure that no one can, say, "hijack" another shopper's shopping cart.

A simple example of forms-based authentication is shown in Listings 4-6, 4-7, and 4-8. This simple-minded example uses a hard-coded user name and password within Login.aspx, as shown in Listing 4-6. These listings also intro-

duce a new class of user interface objects. In Listing 4-6, the button used on the screen isn't a standard HTML submit button or even a standard HTML button but rather an *asp:button*. We'll examine these objects in much greater detail in Chapter 5. For now, just assume that they behave as you might expect. And just take it on faith that the *OnClick* event causes the code in *Login_Click* at the top of the page to be fired. Some of the details within *Login_Click* in Listing 4-6 aren't important, but the call to *FormsAuthentication.RedirectFromLoginPage* is. The first parameter passed to this method is the name of the user, obtained from *UserEmail.Value*, using magic not yet described (See Chapter 5 for more information on getting values from server controls.) The second parameter, hard-coded to *false* here, indicates that a persistent cookie shouldn't be used.

```
<%@ Import Namespace="System.Web.Security " %>

<html>

  <script language="C#" runat=server>
  void Login_Click(Object sender, EventArgs E)
  {
    // Authenticate user: This sample accepts only one user with
    // a name of doug@programmingasp.net and a password of
    // 'password'
    if ((UserEmail.Value == "doug@programmingasp.net") &&
        (UserPass.Value == "password"))
    {
      FormsAuthentication.RedirectFromLoginPage(
        UserEmail.Value, false);
    }
    else
    {
      Msg.Text = "Invalid Credentials: Please try again";
    }
  }
  </script>
  <body>
  <form runat=server>
    <center>
      <h3>
      <font face="Verdana" color=blue>Login Page</font>
      </h3>
      <table>
        <tr>
          <td>
            Email:
          </td>
```

Listing 4-6 A login page for authentication sample (Login.aspx)

(continued)

Listing 4-6 *continued*

```

        <td>
            <input id="UserEmail"
                type="text"
                runat=server
                size=30 />
        </td>
        <td>
            <ASP:RequiredFieldValidator
                ControlToValidate="UserEmail"
                Display="Static" ErrorMessage="*"
                runat=server />
        </td>
    </tr>
    <tr>
        <td>
            Password:
        </td>
        <td>
            <input id="UserPass"
                type=password
                runat=server size=30 />
        </td>
        <td>
            <ASP:RequiredFieldValidator
                ControlToValidate="UserPass"
                Display="Static" ErrorMessage="*"
                runat=server />
        </td>
    </tr>
    <tr>
        <td colspan=3 align="center">
            <asp:button text="Login"
                OnClick="Login_Click"
                runat=server>
            </asp:button>
            <p>
            <asp:Label id="Msg" ForeColor="red"
                Font-Name="Verdana"
                Font-Size="10" runat=server />
            </td>
    </tr>
</table>
</center>
</form>
</body>
</html>

```

Listing 4-7 also shows a pedestrian example (well, pedestrian once you understand how all the ASP.NET form magic works—and you’ll learn all about that in Chapter 5). The form simply identifies the user and allows the user to log out.

```
<%@ Import Namespace="System.Web.Security " %>

<html>

    <script language="C#" runat=server>
    void Page_Load(Object Src, EventArgs E ) {
        Welcome.Text = "Hello, " + User.Identity.Name;
    }

    void Signout_Click(Object sender, EventArgs E) {
        FormsAuthentication.SignOut();
        Response.Redirect("login.aspx");
    }
    </script>

    <body>
        <h3>
            <font face="Verdana">Using Cookie Authentication</font>
        </h3>
        <form runat=server>
            <h3>
                <asp:label id="Welcome" runat=server />
            </h3>
            <asp:button text="Signout" OnClick="Signout_Click" runat=server />
        </form>
    </body>
</html>
```

Listing 4-7 A restricted page for authentication sample that allows you to logout (Default.aspx)

Listing 4-8 is the configuration file for this application, named Web.config. This too is a plain vanilla file. The *authentication* section is the part we’re interested in, and it’s essentially the same as the *authentication* tag shown earlier. Also of interest is the *authorization* tag, which is related to *authentication* as well, as described in the next section.

Note This Web.config file must be at the root of the Web application directory in IIS. Also, the directory must be configured as an application directory, not a virtual directory.

```

<configuration>
  <system.web>
    <authentication mode="Forms">
      <forms name=".ASPXUSERDEMO" loginUrl="login.aspx" protection="All"
        timeout="60" />
    </authentication>
    <authorization>
      <deny users="?" />
    </authorization>
    <globalization requestEncoding="UTF-8" responseEncoding="UTF-8" />
  </system.web>
</configuration>

```

Listing 4-8 Configuration file for authentication sample

There's one more possible twist to forms-based authentication. Within the `<authentication>` tags, a *credentials* section is allowed, where user and password information is allowed. For example, these lines could be added to the *authentication* section of the Web.config file shown in Listing 4-8.

```

<credentials passwordFormat="Clear" >
  <user name="Mary" password="littlelamb"/>
  <user name="Jill" password="uphill"/>
</credentials>

```

The `<credentials>` tag has one attribute, named *passwordFormat*. The possible values for the *passwordFormat* attribute are shown in Table 4-4.

Table 4-4 Options of the *passwordFormat* Attribute

Option	Description
<i>Clear</i>	Stores passwords in clear text. This value is not at all secure, but it is convenient for testing.
<i>SHA1</i>	SHA stands for Secure Hash Algorithm. <i>SHA1</i> stores passwords as SHA1 digests. SHA1 uses a 160-bit hash size. SHA1 was designed to correct a problem in the original SHA algorithm.
<i>MD5</i>	Stores passwords as MD5 digests. <i>MD5</i> produces a 128-bit “fingerprint.” This value is much more reliable than a traditional checksum.

To validate a user name and password from the form, the form needs to call the *Authenticate* method of the *System.Web.Security.FormsAuthentication* class.

The *authorization* Section

After the system has identified a user, you might want to control whether the user is allowed to use the application. The *authorization* section enables you to do exactly that by using `<allow>` and `<deny>` tags, which can specify individual users, or groups of users, called *roles*. Using Windows authentication, as described in the previous section, will cause Windows NT groups to be mapped to roles.

The `<allow>` and `<deny>` tags are searched until the first match is found for the user being authorized. If the first match is in the `<allow>` tag, the user is allowed; if the first match is in the `<deny>` tag, the user is denied. Access is denied if no matching rule is found. In general, for sites where authorization is important, a `<deny users="*" />` tag should be present to make the denial explicit.

The *customErrors* Section

For developers, one of the problems with ASP is a lack of clarity in error messages. ASP.NET has addressed this issue by creating far better error messages, often including not only the single line of code that triggered the error but also a couple lines before and after. This additional information is important because often an error on one line is in fact caused by an error on the previous line. Figure 4-14 shows an example ASP.NET error message.

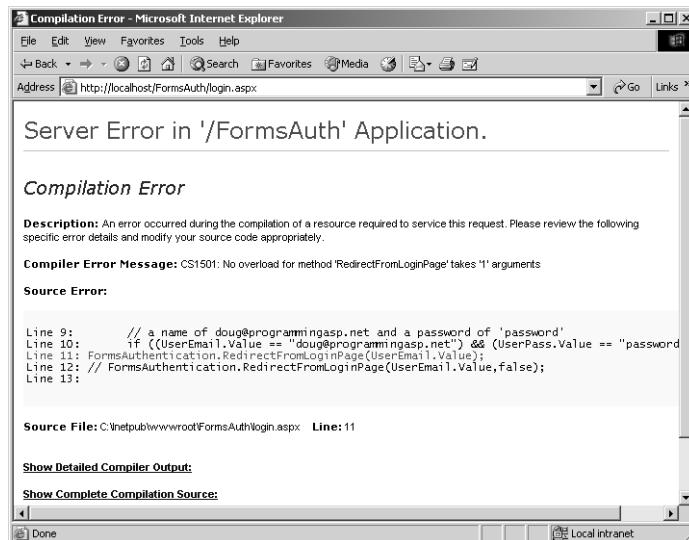


Figure 4-14 An ASP.NET error message

This error page provides a couple links at the bottom that are useful to the developer. The first is Show Detailed Compiler Output. Clicking this link shows the output that would be seen if the command-line compiler were called directly. This output can be useful if there are warnings that occur before the error that might give clues to exactly what's happening. The second link is Show Complete Compilation Source. Clicking this link shows a detailed listing of exactly what the compiler is using to generate the page. The simple Login.aspx page shown in Listing 4-6 is expanded to over 400 lines of detailed listing as ASP.NET takes the source provided (both the code and the HTML source) and produces the code required to create the page. Understanding this code isn't essential, but it can be useful in some debugging situations.

One thing you should notice about the error page is that in showing the context, it actually shows the user name and password that the login page is expecting! Of course, this example is contrived. No one would use such a "security" system in a real application. However, people might have other code that they'd prefer users not see, such as database user names and passwords embedded in connection strings. Embedding connection strings into the application is a bad idea for lots of reasons, but in any event, preventing exposure of source code to users is always a good idea.

The *customErrors* section of Web.config can be used to ensure that this sort of error message appears only to developers during development and testing, and not to users. The `<customErrors>` tag supports the attributes listed in Table 4-5.

Table 4-5 Attributes of the `<customErrors>` Tag

Attribute	Option	Description
<i>defaultRedirect</i> <i>mode</i>		Specifies a URL to redirect the user to
		Specifies whether custom errors are enabled, disabled, or shown only on remote clients
	<i>On</i>	Specifies that custom errors are enabled
	<i>Off</i>	Specifies that custom errors are disabled
	<i>RemoteOnly</i>	Specifies that custom errors are shown only on remote clients

The default behavior for *mode* is *RemoteOnly*, meaning that the type of error page shown in Figure 4-14 will *not* be shown to remote users; instead, a custom error page will appear. The default page shown to remote users is really designed for developers. It explains how to view the details of the error by making changes to the *customErrors* section of Web.config. By setting the *defaultRedirect* attribute, the user can be redirected to a page that can, for example, notify the administrator of the site where an error occurred.

There is also an `<error>` subtag for the `<customErrors>` tag. The `<error>` subtag can occur multiple times. The two attributes in Table 4-6 are supported for the `<error>` subtag.

Table 4-6 Attributes of the `<error>` Subtag

Attribute	Description
<i>statusCode</i>	Specifies an error code that redirects a browser to a nondefault error page
<i>redirect</i>	Specifies the page to redirect to when the error specified in <i>statusCode</i> occurs

The *httpHandlers* Section

The *httpHandlers* section maps incoming requests to the appropriate *IHttpHandler* or *IHttpHandlerFactory* class, according to the URL and the HTTP verb requested.

Note When I talk about *HTTP verbs*, I mean the keywords used to specify what action the Web server should take. If you're an HTML developer, the most common HTTP verbs will be *POST* and *GET*. When you create an HTML form, you have these two options for the *METHOD* attribute of the `<FORM>` tag. When *GET* is specified, any form values are appended to the URL specified in the *ACTION* attribute. When *POST* is specified, form element data is sent as part of the message body. The practical difference is that when *GET* is used, all form elements are sent on the URL, possibly displaying information in the address box that you'd rather not display. (Imagine using *GET* for a login form—the form data, user name, and password will be appended to the URL.) *POST* is preferred, but in practice ASP.NET developers generally allow the framework to handle this type of detail. As you'll see in Chapter 5, most ASP.NET form tags simply use the *runat=server* attribute/value pair.

The *httpHandlers* section in Web.config specifies the HTTP handlers that the applications will use as well as the order in which they will be used. The *httpHandlers* section supports three subtags: `<add>`, `<remove>`, and `<clear>`. The HTTP handler that will be used by an incoming request is determined by looking at all directories that are at a higher level (logically, *not* physically) and processing all the `<add>` and `<remove>` tags. An HTTP module included at a higher level with an `<add>` tag can be removed at a lower level with a `<remove>` tag.

The `<add>` subtag adds an HTTP handler, and it supports the three attributes listed in Table 4-7.

Table 4-7 Attributes of the `<add>` Subtag in `httpHandlers`

Attribute	Description
<i>verb</i>	A comma-separated list of HTTP verbs, such as <i>GET</i> , <i>PUT</i> , <i>POST</i> , or the wildcard character (*).
<i>path</i>	A single URL path or a simple wildcard, such as <i>*.aspx</i> .
<i>type</i>	An assembly and class combination. Assemblies are groups of functionality grouped together for convenience. The .NET Framework first searches in the application's private bin directory and then in the system assembly cache.

The `<remove>` subtag removes an HTTP handler specified previously in an `<add>` subtag. The *verb/path* in a `<remove>` subtag must *exactly* match the *verb/path* specified in a previous `<add>` subtag. Although it might seem silly to add and remove HTTP handlers, remember that the configuration files are read from the root through to the current directory (logically speaking, *not* physically), so it's not unreasonable to expect that there will be times when a handler needed at a higher level isn't needed in every application located logically below. The `<remove>` subtag supports two attributes, *verb* and *path*. These are exactly the same as their `<add>` subtag counterparts.

The final subtag supported by the `httpHandlers` section is the `<clear>` subtag. When this subtag is present, all HTTP handler mappings inherited or configured are cleared. Here is a simple example of an `httpHandlers` section:

```
<httpHandlers>
  <add verb="*" path="MyApp.New" type="MyApp.New, MyApp" />
  <add verb="*" path="MyApp.Baz" type="MyApp.Baz, MyApp" />
</httpHandlers>
```

In this example, all HTTP verbs that are used on *MyApp.New* will be mapped to *MyApp.New* class in the *MyApp* assembly. All HTTP verbs used on *MyApp.Baz* will be referred to *MyApp.Baz* class, in the *MyApp* assembly.

The `httpModules` Section

The `httpModules` section of `Web.config` contains information similar to the `httpHandlers` section described in the preceding section. The `httpModules` section supports three subtags, `<add>`, `<remove>`, and `<clear>`, just as in the `httpHandlers` section. The `<add>` subtag supports two attributes, *type* and *name*, as described in Table 4-8.

Table 4-8 Attributes of the <add> Subtag in *httpModules*

Attribute	Description
<i>type</i>	Specifies a comma-separated class/assembly combination. ASP.NET searches the application's private bin directory and then the system assembly cache.
<i>name</i>	Specifies the name that the application uses to refer to the module identified in <i>type</i> .

The <remove> subtag works exactly the same as the <remove> subtag in the *httpHandlers* section. The *type* and *name* attributes are used to match previously added HTTP modules. The <clear> subtag removes all HTTP module mappings from an application.

The *identity* Section

The *identity* section of Web.config controls the application identity of the Web application. This section allows you to set up *impersonation*. Impersonation is when the identity of the user on the client machine is used to determine what files on the server can be accessed. For example, suppose you have two virtual directories on an intranet-accessible Web server, one named Employees and the other named Managers. If all users are using Windows and have Windows 2000 domain user accounts and the Web server has both directories on an NTFS volume, instead of using application logic to prevent nonmanagers from accessing the Managers virtual directory, you could apply NTFS permissions to the files in the Managers directory that allowed only managers to access the files in that directory.

The <identity> tag supports three attributes, as described in Table 4-9.

Table 4-9 Attributes of the <identity> Tag

Attribute	Option	Description
<i>impersonate</i>		Specifies whether client impersonation is used on each request
	<i>true</i>	Specifies that client impersonation is used
	<i>false</i>	Specifies that client impersonation is not used, which is the default
<i>userName</i>		Specifies the user name to use if <i>impersonate</i> is set to <i>false</i>
<i>password</i>		Specifies the password to use if <i>impersonate</i> is set to <i>false</i>

The *pages* Section

The *pages* section of the Web.config file contains page-specific information that can be configured on the machine, site, application, or virtual directory level. The `<pages>` tag supports six attributes, as listed in Table 4-10.

Table 4-10 Attributes of the `<pages>` Tag

Attribute	Option	Description
<i>buffer</i>		Specifies whether the URL resource uses response buffering
	<i>true</i>	Specifies that response buffering is enabled
	<i>false</i>	Specifies that response buffering is disabled
<i>enableSessionState</i>		Specifies whether session state is enabled
	<i>true</i>	Specifies that session state is enabled
	<i>false</i>	Specifies that session state is disabled
	<i>ReadOnly</i>	Specifies that session state data can be read but not written
<i>enableViewState</i>		Specifies whether view state (the state of controls) is enabled
	<i>true</i>	Specifies that view state is enabled
	<i>False</i>	Specifies that view state is disabled
<i>pageBaseType</i>		Specifies a code-behind class that .aspx pages inherit
<i>userControlBaseType</i>		Specifies a user control that user controls inherit
<i>autoEventWireup</i>		Indicates whether page events are automatically enabled
	<i>true</i>	Indicates that page events are automatically enabled
	<i>false</i>	Indicates that page events are not automatically enabled

Note The *autoEventWireup* attribute seemed like a good idea at the time. Events are wired up based on the names of methods and components. Given a control named *button1*, *button1_Click* would handle the click event. In general, the attribute is more trouble than it's worth and has been the cause of no end of confusion on the ASP.NET newsgroups. All the examples in this book will use manual event *wireup*.



ASP.NET Differences ASP developers are used to the fact that if you move beyond a single server, session state can't be saved using the standard ASP session state mechanisms. The problem is that in ASP, session state is stored directly on the Web server. In a clustered Web server farm environment, there's no assurance that each request from a particular client will go to a particular server in the cluster. Workarounds in ASP include saving small bits of session state in encrypted cookies and then using that little bit of session state to go to a database to get other information. Another workaround is to use a session identifier that gets passed from page to page and use that to get to a database. ASP.NET eliminates the need for these workarounds. Session state can be saved in a state server or SQL Server database. You'll find more information on session state in the section "The *sessionState* Section" later in this chapter.

The *processModel* Section

The *processModel* section of Web.config controls the ASP.NET process model settings on an IIS Web server. This section is different from the other sections we've looked at in that it's read by the aspnet_isapi unmanaged DLL rather than the managed code configuration system. The *processModel* section sets many performance-tuning details.

Caution Many features can be configured using the *processModel* section, including attributes that specify how ASP.NET runs on multiprocessor machines. You can do things such as set the CPU mask, meaning that you can control which processors ASP.NET will use to execute its code. If this sounds like a good idea, think again. Microsoft has spent perhaps millions of dollars and untold hours creating the Windows 2000 process scheduling system, which efficiently handles doling out works to the multiple processors in a multiple CPU system. Only under unique circumstances are you likely to do better manually mucking with setting processor affinity.

The `<processModel>` tag supports many attributes. The most common ones are described in Table 4-11.

ASP.NET runs one process per eligible CPU. On a four-processor system, if all CPUs are eligible to run ASP.NET (based on the settings for the *cpuMask* and *webGarden* attributes), four ASP.NET processes will be started. Given a *cpuMask* of 7 (as described earlier), only three ASP.NET processes will be created.

Table 4-11 Attributes of the <processModel> Tag

Attribute	Option	Description
<i>enable</i>		Specifies whether the process model is enabled.
	<i>true</i>	Indicates that the process model is enabled.
	<i>false</i>	Indicates that the process model is not enabled.
<i>timeout</i>		Specifies the number of minutes until ASP.NET launches a new worker process to take the place of the current one. The default is <i>infinite</i> .
<i>idleTimeout</i>		Specifies the number of minutes of inactivity until ASP.NET automatically shuts down the worker process. The default is <i>infinite</i> .
<i>shutdownTimeout</i>		Specifies the number of minutes allowed for a worker process to shut itself down. If the timeout expires and the worker process hasn't shut itself down, ASP.NET shuts down the process. The format is hr:min:sec. The default is 5 seconds, or 0:00:05.
<i>requestLimit</i>		Specifies the number of requests allowed before ASP.NET automatically launches a new worker process to take the place of the current one. The default is infinite.
<i>requestQueueLimit</i>		Specifies the number of requests allowed in the queue before ASP.NET launches a new worker process and reassigns the requests. The default is 5000.
<i>memoryLimit</i>		Specifies the maximum memory size, as a percentage of total system memory, that the worker process can consume before ASP.NET launches a new process and reassigns existing requests. The default is 40. A percent sign (%) is <i>not</i> specified; only the number.
<i>cpuMask</i>		Specifies a bitmask value that indicates which processors in a multiple-processor system are eligible to run ASP.NET processes. On a computer with four CPUs, a value of 0111 binary (7 decimal) would mean that CPUs 0 through 2 would run an ASP.NET process and CPU 3 would not. This attribute interacts with the <i>webGarden</i> attribute.

Table 4-11 *continued*

Attribute	Option	Description
<i>webGarden</i>		Controls CPU affinity when used in conjunction with the <i>cpuMask</i> attribute. A multiple-processor system is called a Web garden, presumably in contrast to a multiple-PC cluster, often called a Web server farm.
	<i>true</i>	Specifies that the system should use the Windows CPU scheduling system. This is the default.
	<i>false</i>	Specifies that <i>cpuMask</i> is used to specify which CPUs are eligible to run ASP.NET processes.
<i>userName</i>		Specifies an account that worker processes should use. By default, processes run using the IIS account.
<i>password</i>		Specifies the password for the account specified in the <i>username</i> attribute.
<i>logLevel</i>		Specifies event types logged to the event log.
	<i>All</i>	Specifies all process events are logged.
	<i>None</i>	Specifies no process events are logged.
	<i>Errors</i>	Specifies only errors are logged. These include unexpected shutdowns, memory limit shutdowns, and deadlock shutdowns. <i>Errors</i> is the default.
<i>clientConnectedCheck</i>		Specifies the time a request is left in the queue before a check is made to see if client is still connected.
<i>comAuthenticationLevel</i>		Specifies the level of authentication for DCOM security. The options for this attribute are: <i>Default</i> , <i>None</i> , <i>Connect</i> , <i>Call</i> , <i>Pkt</i> , <i>PktIntegrity</i> , and <i>PktPrivacy</i> . The default is <i>Connect</i> .
<i>comImpersonationLevel</i>		Specifies the authentication level for COM security. The options for this attribute are: <i>Default</i> , <i>Anonymous</i> , <i>Identify</i> , <i>Impersonate</i> , and <i>Delegate</i> . (<i>Anonymous</i> is currently not supported.)
<i>maxWorkerThreads</i>	5 to 100	Specifies the maximum number of worker threads to be used for the process on a per CPU basis. The default is 25.
<i>maxIoThreads</i>	5 to 100	Specifies the maximum number of I/O threads to used for the process on a per CPU basis. The default is 25.

The *sessionState* Section

Session state support in ASP.NET is much more extensive and flexible than it was in ASP. For developers of small Internet or intranet Web sites, the session support offered by ASP was adequate. The problem was that ASP session state didn't scale out to multiple Web servers. ASP session state was stored on the Web server, and so using a system like Microsoft's Network Load Balancing provided no assurance that the same server in a Web server farm would service each request from a particular client. Another limitation of ASP session state is that it requires cookies to work. This constraint has become less of a problem because now virtually all browsers support cookies, and the sheer number of Internet sites that require cookies enabled have forced all but the most paranoid users to accept at least nonpersistent cookies.

The *sessionState* section of Web.config controls how session state is managed. The `<sessionState>` tag supports five attributes, as described in Table 4-12.

Table 4-12 Attributes of the `<sessionState>` Tag

Attribute	Option	Description
<i>mode</i>		Specifies where session state is stored.
	<i>Off</i>	Specifies that no session state is saved.
	<i>Inproc</i>	Specifies that session state is saved locally, similar to ASP session state.
	<i>StateServer</i>	Specifies that session state is saved on a remote state server.
	<i>SqlServer</i>	Specifies that session state is saved in a SQL Server.
<i>cookieless</i>		Specifies whether session state should be saved without using client cookies.
	<i>true</i>	Specifies that sessions without cookies are being used.
	<i>false</i>	Specifies that sessions do use cookies. This is the default.
<i>timeout</i>		Specifies the number of minutes a session can be idle before it is abandoned. The default is 20 minutes, the same as in ASP.

Table 4-12 *continued*

Attribute	Option	Description
<i>stateConnectionString</i>		Specifies the server name and port where session state is stored remotely (for example, <i>192.168.1.100:8484</i>). This attribute is required when mode is set to <i>StateServer</i> .
<i>sqlConnectionString</i>		Specifies the connection string for the SQL Server where the state is to be saved (for example, <i>data source=192.168.1.100;user id=sa;password=</i>). This attribute is required when mode is set to <i>SqlServer</i> .

The same rules about trying to minimize the amount of data stored in session state that was applied in ASP still apply in ASP.NET.

The *trace* Section

One of the problems developers experienced with ASP was difficulty obtaining detailed debugging information. Exactly what was the page doing when an error occurred? What portions of the code had been run? ASP.NET has much improved debugging information, and the *trace* section of the Web.config file allows you to specify settings for the trace service. Figure 4-15 shows a page that has been run with tracing enabled and *pageOutput* set to *true*.

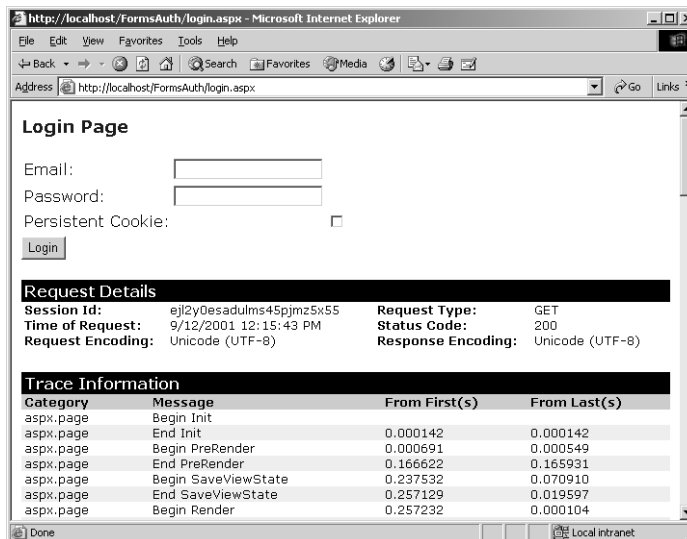


Figure 4-15 The output of the Login.aspx page shown in Listing 4-6 when tracing is enabled and *pageOutput* is set to *true*

The `<trace>` tag supports five attributes, as described in Table 4-13.

Table 4-13 Attributes of the `<trace>` Tag

Attribute	Option	Description
<i>enabled</i>		Specifies whether tracing is enabled.
	<i>true</i>	Specifies that tracing is enabled.
	<i>false</i>	Specifies that tracing is disabled. This is the default.
<i>requestLimit</i>		Specifies the number of trace requests to save on the server. The default is <i>10</i> .
<i>pageOutput</i>		Specifies whether trace information should be displayed at the end of each page.
	<i>true</i>	Specifies that trace output is appended to each page.
	<i>false</i>	Specifies that trace output is not appended to each page. This is the default.
<i>traceMode</i>		Sets the order of trace output.
	<i>SortByTime</i>	Specifies that output is sorted by time (thus, displayed in the order the events being traced occurred). This is the default.
	<i>SortByCategory</i>	Specifies that output is displayed alphabetically by category. See the text in this section for information about user-defined categories.
<i>localOnly</i>		Specifies whether the trace viewer is available only on the host Web server.
	<i>true</i>	Specifies that trace output is available only on the server console. This is the default.
	<i>false</i>	Specifies that trace output is available on any client, not just the Web server.

In Figure 4-15, all trace output has a category of *aspx.page*, and it is generated automatically by the .NET Framework, with no explicit trace code in the page source. However, this is only half the power of ASP.NET tracing. Suppose, for example, that the login page from Listing 4-6 wasn't responding the way you expected. Perhaps the following code seemed not to be working correctly:

```
if ((UserEmail.Value == "doug@programmingasp.net") &&
    (UserPass.Value == "password"))
{
    FormsAuthentication.RedirectFromLoginPage(
        UserEmail.Value, false);
}
```

```

}
else
{
    Msg.Text = "Invalid Credentials: Please try again";
}

```

You can use the *Trace* class to add user-defined trace statements in the trace output, like this:

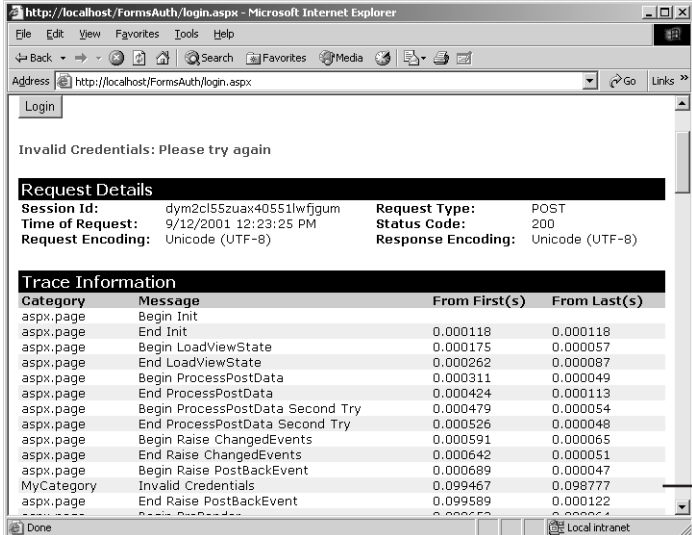
```

if ((UserEmail.Value == "doug@programmingasp.net") &&
    (UserPass.Value == "password"))
{
    Trace.Write("MyCategory", "Authenticated");
    FormsAuthentication.RedirectFromLoginPage(
        UserEmail.Value, false);
}
else
{
    Msg.Text = "Invalid Credentials: Please try again";
    Trace.Write("MyCategory", "Invalid Credentials");
}

```

If you ran the page shown in Listing 4-6 with the modifications shown above and entered an invalid user name or password, the trace would have one “MyCategory” trace line inserted into the output, as shown in Figure 4-16.

In addition to the Web.config sections described in this chapter, there are a couple other sections that are not generally modified (such as the *globalization* section).



The screenshot shows a web browser window with the address `http://localhost/FormsAuth/login.aspx`. The page displays a login form with a "Login" button and an error message: "Invalid Credentials: Please try again". Below the error message, there is a "Request Details" section and a "Trace Information" section.

Request Details

Session Id:	dym2cl55zuax40551lwfgum	Request Type:	POST
Time of Request:	9/12/2001 12:23:25 PM	Status Code:	200
Request Encoding:	Unicode (UTF-8)	Response Encoding:	Unicode (UTF-8)

Trace Information

Category	Message	From First(s)	From Last(s)
aspx.page	Begin Init		
aspx.page	End Init	0.000118	0.000118
aspx.page	Begin LoadViewState	0.000175	0.000057
aspx.page	End LoadViewState	0.000262	0.000087
aspx.page	Begin ProcessPostData	0.000311	0.000049
aspx.page	End ProcessPostData	0.000424	0.000113
aspx.page	Begin ProcessPostData Second Try	0.000479	0.000054
aspx.page	End ProcessPostData Second Try	0.000526	0.000048
aspx.page	Begin Raise ChangedEvents	0.000591	0.000065
aspx.page	End Raise ChangedEvents	0.000642	0.000051
aspx.page	Begin RaisePostBackEvent	0.000689	0.000047
MyCategory	Invalid Credentials	0.099467	0.098777
aspx.page	End RaisePostBackEvent	0.099589	0.000122

A callout points to the "MyCategory" trace line in the trace information table, labeled "User-defined trace output".

Figure 4-16 Trace output with explicit trace category added

Conclusion

One important thing to keep in mind about these configuration settings is that in many cases, the default values are completely acceptable. In the real world, most of the sections described in this chapter won't exist in your Web.config files because the default values will suffice. In addition, tools such as Visual Studio .NET will interact with some of these settings on their own; thus, the default behavior you see if you're using Visual Studio .NET might be slightly different from what has been described as the ASP.NET default. Users of Visual Studio .NET will notice other significant differences compared to the experience of someone using an editor that isn't at all .NET aware. In general, most of the examples in the following chapters use Visual Studio .NET, but with special attention paid to the "magic" that Visual Studio .NET performs on your behalf. Generally, Visual Studio .NET is doing things that are helpful, but understanding what is going on "behind the curtain" will help on those occasions when the special Visual Studio .NET behavior isn't what you're looking for.

With this background, you should know enough to move on to Chapter 5, which covers the most important type of ASP.NET application—the Web Form. Web Forms provide ASP.NET developers with the kind of rapid application development (RAD) for the server that ASP programmers could only dream of. The dream has arrived, and in Chapter 5 you'll see how to make it work for you!